How to Think About Security Failures

Understanding complexity and feedback in security models highlights the need for better failure modes in solutions.

Why is securing large computer systems so difficult? It's not just for the obvious reasons—that they're large and publicly available through the Internet. Nor is the problem new, unstudied, or without considerable motivation among researchers in government laboratories and in commercial enterprises. Defenders build the virtual walls of computing centers thicker and higher, while attackers exercise brilliantly diabolical attacks on weak links, turning systems against themselves in an arms race that has gone on for as long as computers have communicated through networks.

So rather than suggest a technological solution or some clever new way of identifying attackers—of which there are many fascinating and effective examples—I examine some basic problems involving human intuition and how it can influence the security design of complex systems. By focusing not only on understanding how we create and use models to solve problems, but on how these solutions sometimes fail, it is possible to develop ways to think about security problems that result in significantly more robust solutions.

Intuition can be viewed (roughly) as a model for familiar events. In models, assumptions simplify the data being looked at to a degree that's proportional to the generality of the model. Simplification is both necessary and built into what makes models useful for problem solving. As the models provide solutions for classes of problems, we develop intuition that encapsulates our familiarity with them. Unfortunately, along with familiarity comes the tendency to miss changes that are inconsistent with the assumptions we made when creating the models. When this happens, a model may end up causing more harm than good by coloring how we analyze its results.

What does this have to do with computer security? If the basic assumptions behind the intuition are inconsistent with the actual behavior or nature of the threat, the system may be less protected than we think it is, as illustrated by the following example: In a classic (circa 1995) network security design, your opponent is usually viewed as a hostile agent from outside the network attempting to subvert defenses via weakness in accessible services. If a host or service within the network cannot be touched from the outside, it is considered safe from external attack. To protect users, no remote host is allowed to connect to their workstations. Today, a new class of attack—the client-side attack—has become increasingly popular and dangerous; examples

LISA HANEY

Even if you miss the rock falling into the pool, you might catch sight of the ripples moving on the surface.

include hostile email attachments and Web pages that allow arbitrary code to be run in light of security problems in the Web browser or email client. With the rise of client-side attacks, a flaw emerges in the old model; despite avoiding a direct connection to the outside, users might still be attacked by the very services they've requested [3].

A new attack vector has been created in which users are transformed into a platform to attack internal resources without their consent or even their awareness. Users are no longer passive participants in the security model; they've become the very service by which entrance is gained into the protected interior of the network. The point is that the environment in which a model lives is constantly changing. If this change is ignored, the model inevitably fails.

Unexpected Consequences

Compounding the problem of continuing changes in network and software behavior is the fact that real-world collections of software, hardware, and people often act and fail in ways that are difficult or impossible to predict. While large groups of people behave in a statistically predictable manner, the effects of simple changes to an otherwise well-ordered system do not always result in predictable results. The exact problem you want solved may be solved, even as the overall result fails.

This outcome is embodied in the notion of unexpected consequences. An example involves user authentication, an old friend of computer security. For as long as people have used passwords for authentication, other people have been interested in stealing them (credential theft). In order to combat credential theft, commercial technology introduced over the past several years creates single-use (one-time) passwords that replace the traditional idea of a password. The technology being used to do this is irrelevant, except in the sense that the credential for authenticating a user can be used only one time.

One assumption is that neither side of the communication channel has been altered. Sadly, in a large number of cases of deliberate credential theft a user logs in from a machine where one side of the channel has been compromised. For example, one unsuspecting user, Bob, authenticates and logs in to Alice's cluster from his workstation using his one-time password. Unfortunately, a hostile party, Eve, has taken over that workstation without Bob's knowledge. Bob now begins to work on Alice's cluster. Eve can now use this authenticated connection to attack Alice's cluster without Bob knowing that something is happening behind the scenes. The actual mechanism for doing this is not technically complex, and many toolkits provide Eve the needed functionality without much effort on her part [1, 4].

Considering how a securely authenticated link can be used to attack Alice's cluster, it's important to note that the problem of credential theft is completely solved; Eve has access to the authentication information that Bob types in, even though it is useless after the login. By viewing Bob as a service to access Alice's cluster rather than as a source of authentication tokens, Eve has managed to get around Alice's mechanism for preventing illegal

access. Further, unless Alice is prepared to deal with this new threat, she will face an attack far more subtle and dangerous than simple credential theft (which she has years of experience detecting and correcting).

Evolving Ecology

What these examples of security problems have in common is that in resolving a specific vulnerability, something more complicated took its place. The act of removing a vulnerability puts pressure on the entire ecology around the system. Rather than just disappearing, attackers eagerly evolve to take advantage of whatever opportunities remain. Securing a system exerts pressure on attackers, forcing changes in their formerly successful behavior, along with the creation of novel and previously unexplored attack patterns to take advantage of still unprotected resources. These resources—vulnerabilities and unexplored options—have always been there, it's just that there hasn't been sufficient motivation to use them, since there were always simpler ways to get by.

Security design must recognize that enhancements are the driving force behind new intrusion techniques, rather than simply being a means of removing vulnerabilities. The importance of this point cannot be overemphasized.

How does this relate to my earlier discussion of models? If Alice tries to protect her system administrators by using a firewall to isolate them from contact initiated from the outside, Eve may make short work of Alice's defenses by sending an email message that directs one of their vulnerable Web browsers to a hostile Internet site. Alice knows that while there is utility in blocking connectivity, it is equally important to screen email and Web content and keep workstation and application software patched.

Resolution

How could this insight help design more secure systems? First, the models and assumptions used to develop security solutions must be grounded in real-world data and account for the possibility of failure due to unexpected behavior, both human and technological. In addition, resolving one security vulnerability might cause changes (often significant and unpredictable) in the distribution of attack methods.

Any design will fail at some point [2]. However, if you design with the inevitability of failure in mind, when it happens you'll at least have a chance to find out about it. The key is designing systems that are able to fail gracefully. Determining that there is a problem when it happens is the best option for minimizing damage, besides preventing it outright. Solutions must be designed to make a great deal of noise when they fail or misbehave. Most systems end up doing something unexpected. When they do, you'll want to know about it. Even if you miss the rock falling into the pool, you might at least catch sight of the ripples moving on the surface.

So, what should Alice do to augment her new one-time password software? She knows that Eve will not relent, so assuming that the authentication system is likely to fail will allow for many more options. Perhaps the communication application Bob uses to log in is now instrumented to audit user activity. Perhaps system accounting provides feedback for unusual behavior. This may not stop the attack, but it will be possible to detect and respond when the unexpected inevitably happens. C

References

1. RD (rd@thehackerschoice.com). Writing Linux Kernel Keylogger. *Phrack 11,* 59 (June 2002); www.phrack.org/phrack/59/p59-0x0e.txt.
2. Schneier, B. *Secrets & Lies: Digital Security in a Networked World.* Wiley Computer Publishing, New York, 2000.
3. Wang, Y.-M. Strider HoneyMonkeys: Active client-side honeypots for finding Web sites that exploit browser vulnerabilities. Part of Works in Progress at the 14th Usenix Security Symposium (Baltimore, July 31–Aug. 5, 2005); www.usenix.org/events/sec05/wips/wang.pdf and research.microsoft.com/HoneyMonkey/.
4. Melstorm (melstorm@storm.net.nz). Trust Transience: Post-Intrusion SSH Hijacking Melstorm. Presentation at Defcon 14 (Las Vegas, Aug. 4–6, 2005); opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-05/BH_US_05-Boileau/metlstorms_sshjack-1.02.tar.gz.

Scott Campbell (scampbell@lbl.gov) is a computer security analyst at the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory, Berkeley, CA.